Nix and Open Source Software for the Arts

Abstract — Functional programming is a subset of programming based upon Lambda calculus, it treats programming in a mathematical sense where the fundamental building block of the program are functions and data is immutable. Nix is a purely functional package manager used to create reproducible binaries and development environments, it treats packages, and program dependencies as immutable values that are built by pure functions and controlled by a lock file. Nix leverages **nixpkgs**, an open source collection of over 120,000 software packages that can be installed with the Nix, making its GitHub repository on the largest and most active in the world. In this work we will be reviewing nixpkgs and how open source hardware and software relates to the arts, as well as evaluating the technological sustainability of such open source repositories and how community driven maintenance can affect the life cycle of a nix Package.

I. An introduction to functional programming

Functional programming is a programming paradigm, whose basal unit of computation is a function, a self contained modules of code that accomplish a specific task [1]). Complex programs are then built through the composition and chaining of these functions. There a few a core tenets to the functional paradigm, the main two being **immutability** and **functional purity**, both of which are intrinsically linked. The concept of functional purity is that given the same inputs, a given functional will always produce the same outputs. Pure functions are said to have referential transparency. A classic example of would a function which given any number n, adds one to it and returns the result $f(x) = x + f$. This function, when given the input 3 always produces 4. In a programming language such as Haskell, it would be written as:

```haskell
addOne :: Int -> Int
addOne x = x + 1
```

Figure 1 - A function which always adds one to a number in Haskell

However, software is rarely pure, that is to say, that there many conditions in which the output of a function can change. For instance if a function requires a connection to a database or to read a file then connection can fail and the file could not be present. Both of these conditions elements outside our code which can have an affect results a function which relies on said database or file. This ambiguity is referred to as a side effect, and functions that perform them are deemed impure. In purely functional languages, even mutating a variable is considered a side effect, as calling a function which uses said mutable variable as an argument multiple times may yield different outputs.

Referential Transparency The ability to replace a function with its results without affecting the surrounding program

The package manger and build system, Nix, is purely functional and operates under this same functional programming maxim of immutability. In Nix the packages used to build development environments and software are immutable, and never change after being built. Nix stores packages in a Nix store, where each package has its own unique directory denoted by a cryptographic hash of the package's build dependency graph [2]. Nix (the package manger) configured in Nix (the functional programming language), where packages are built from *Nix expressions*. A Nix expression describes all aspects of a package build action to from a derivation. Nix expressions are deterministic, as in that building a package from an Nix expression twice will produce the same output.

II. Nix the functional programming language

Nix a domain specific purely functional, dynamically type and lazily evaluated programming language. Nix expressions are expressions written in the Nix programming language, a language with a JSON-esque syntax. Nix even describes itself as JSON with functions. Values in the Nix programming language can be either primitive data type, lists, attribute sets and functions [3]. An attribute set in Nix is a collection of nae value pairs, similar to dictionaries or hash maps in other languages.

```nix
{
  string = "hello";
  integer = 1;
  float = 3.141;
  bool = true;
  null = null;
  list = [ 1 "two" false ];
  attribute-set = {
    a = "hello";
    b = 2;
    c = 2.718;
    d = false;
  }; # comments are supported
}
```

Figure 2. - An attribute set in the Nix programming language

Being a functional programming language, functions are at the core of nix code. In nix functions always take exactly one argument and the argument and function body are separated by a `:`, where the left side is function argument and the right the body. All functions in nix are lambda expressions.

```nix
{ a, b }: a + b
```

Figure 3. - A Nix function that takes an attribute set as an argument

Build inputs in Nix are specified explicitly in two ways, file system paths or through dedicated functions. In Nix, whenever a system path is used via string interpolation, the contents of the file in that path are copied to Nix's own filesystem abstraction, the **nix store**. In nix, build inputs do not have to come from the local file system.As part of the Nix's standard library, built in impure functions are provided to fetch files over the network.

- `builtins.fetchurl`
- `builtins.fetchTarball`
- `builtins.fetchGit`
- `builtins.fetchClosure`

Naturally an error occurs if the network request fails, or the if the file is not present at its specified location when using local inputs.

Build Inputs Build inputs are files that derivations refer to in order to describe how to derive new files. When run, a derivation will only have access to explicitly declared build inputs.

III. Nix the build system

Build systems all serve one fundamental purpose, to transform source code into executable binaries. While for simple projects, simply evoking the source language's compiler from the terminal with commands such as `cargo build` for Rust or `cabal build` for Haskell may be sufficient, but as the project scope increases complication arise and the need for a dedicated build system becomes more apparent. And for projects where the source code is multiple languages and/or compilation units building is no longer a single step process. A build system not only manages compilation of the project, but it also evaluates the projects dependencies so that it does not rely on any stale binaries.

Stale binaries A compiled program or component file which is out of date with respect to the rest of the source code or the dependencies which is built upon. Stale binaries can lead to incorrect behaviour and hard to trace bugs in a codebase.

Build systems are designed to be both correct and fast, achieving this largely through the use of robust dependency graphs defining the inputs for every build target in the project (source files, configuration, tools) and clever hashing of file contents such that even if a files time stamp has changed, if the file remains genuinely the same, then re-compilation does not need to occur for that file, greatly reducing compile times.

Nix is as much a build system as it is a package manger. Nix can build software form source and distribute it as package. A Nix package is a expression in the Nix programming language which will evaluate to a derivation, a sequence of build steps needed to generate the data required to make a given piece of software. Nix expressions describe how to build packages from source and collected in the *nixpkgs* repository. Using these expressions the nix package manger can build binary packages, as well a development environments known as Nix-shells. Nix has uses its own immutable abstraction of the file system known as a *Nix store*.

The default nix store on the local files systems is `/nix/store`. Inside a nix store there are a collection of *store objects*, a store object can consist of:

- A file system object as data
- A set of store paths as references to store objects

A file system object will be one of the following:

- A file
- A Directory
- A Symbolic Link (A file which refers to another file in another directory as a path)

A store path is an opaque (as in a datatype who's that internal structure is not available), unique identifier. A store path will always reference exactly one store object. Store objects are pairs of:

1. A 20-byte digest for identification
2. A human readable symbolic name

Store objects are immutable, as in that once they are created, they do not change nor can any other store object that they reference can be changed. The store plays a pivotal role in way Nix creates derivations for building software, as derivations themselves are specifications for running a given executable on the specified input.

Nix Derivations A specification for running an executable on precisely defined input to produce on more store objects. These store objects are known as the derivation's *outputs*. Derivations are *built*, in which case the process is spawned according to the spec, and when it exits, required to leave behind files which will (after post-processing) become the outputs of the derivation.

Naturally Nix derivations can be made in conjunction with language specific package mangers such as Rust's `cargo`. With many open source projects such as *poetry2nix*, *uv2nix* and *cabal2nix* capable of creating nix derivations based upon their languages respective package mangers. For languages such as Rust, Nix offers first class support for making derivations based upon a rust projects `Cargo.toml` and `Cargo.lock` files.

```
manifest = (pkgs-stable.lib.importTOML ./
Cargo.toml).package;

rustPackage = pkgs-
stable.rustPlatform.buildRustPackage {
    inherit (manifest) name version description;
    cargoLock.lockFile = ./Cargo.lock;
    src = pkgs-stable.lib.cleanSource ./.;
};
```

Figure 4. - Parsing Rust `Cargo.toml` with the nix function `pkgs-stable.lib.importTOML` and then reading the rust source code dependencies using the function `pkgs-stable.rustPlatform.buildRustPackage`. Both of these functions are part of Nix's standard library.

The `rustPlatform.buildRustPackage` function takes the source code from `src` and the projects `Cargo.lock` file and builds the project as via `Cargo` using the Rust toolchain already available no nixpkgs.

IV. Nix the package manager

A package manger is a tool that automates code reuse. They ensuring that all third party dependencies needed for a project, either that is directly or indirectly are downloaded and version controlled in manner which no dependencies conflict [4]. Package mangers usually operate at two levels, either tied to a programming language like the package manger `cabal` is tied to Haskell, where they are tasked with finding dependencies related to a projects source code, or at a system level managing full software installation such as `brew` on MacOS.

The Nix package manager operates on the system level, and leverages *nixpkgs*, a software distribution built with Nix (the programming language), it contains 120,000 packages all released under the various open source licenses. These packages can then be used with the Nix package manager on most Linux distributions. Packages are distributed channels, for use on non NixOs Linux distributions, package distribution is done through the `nixpkgs-unstable` channel. Both nixos-unstable and nixpkgs-stable follow the master branch of the nixpkgs repository.

Nix Channels A pointer to versioned tarballs and git commits.

Nix supports a verity of platforms with various amounts of support for each:

1. `x86_64-linux`: Highest level of support.
2. `aarch64-linux`: Well supported, with most packages building successfully in CI.
3. `aarch64-darwin`: Receives better support than `x86_64-darwin`.
4. `x86_64-darwin`: Receives some support.

Nix packages can be modified and extended via *Overlays*. Overlays are Nix functions which accept two arguments (idiomatically referred to as `final` and `prev`) and return a set of packages. They are similar to the `packageOverrides` function, but more flexible, as they are capable of taking in arguments beyond `final` and `prev`. Overlays are commonly used to either patch, or pin package to a version that nixpkgs does not ship. An example of which is the following:

```
final: prev:
{
    sl = prev.sl.overrideAttrs (old: {
        src = prev.fetchFromGitHub {
            owner = "mtoyoda";
            repo  = "sl";
            rev   =
"923e7d7ebc5c1f009755bdeb789ac25658ccce03";
            hash  =
"173gxk0ymiw94glyjzjizp8bv8g72gwkjhacigd1an09jshdrjb4";
        };
    });
}
```

Figure 5 - Example nix expression pinning the `sl` package specifically to the `923e7d7ebc5c1f009755bdeb789ac25658ccce03` revision. Placeholder hashes such as `sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=` can be used, and Nix's `hash mismatch error` will provide the correct one as part of its error message upon evaluation of the nix expression.

It is similarly easy to add patches to a package:

```
final: prev:
{
  sl = prev.sl.overrideAttrs (old: {
    patches = (old.patches or []) ++ [
      (prev.fetchpatch {
        url = "https://github.com/charlieLehman/sl/
commit/e20abbd7e1ee26af53f34451a8f7ad79b27a4c0a.
patch";
        hash =
"07sx98d422589gxr8wflfpkdd0k44kbagxl3b51i56ky2wfix7rc";
      })
      # alternatively if you have a local patch,
      /path/to/file.patch
      # or a relative path (relative to the current
nix file)
      ./relative.patch
    ];
  });
```

Figure 6. - A patching a package in Nix. This is done my overriding the source `src` of the target package with the patched version [5]

By building software as a nix package, nix also becomes a means for software distribution. The command `nix run` allows for the allows for the execution of applications or binaries provided by Nix packages or flakes. Making it a streamlined way to run software without installing it system-wide. The command: `nix run blender-bin`. Runs the default app from the `blender-bin` flake. To use nix run the flake must evaluate to an app or regular nix derivation. If it evaluates to an app, then `nix run` executes the program specified by the app definition.

```
apps.x86_64-linux.blender_2_79 = {
    type    = "app";
    program = "${self.packages.x86_64-
linux.blender_2_79}/bin/blender";
};
```

Figure 7. - A nix app definition. Here the target system is specified with `apps.x86_64-linux`

If `nix run` evaluates to a derivation, then nix will try to execute the program `<out>/bin/<name>`, in which `out` refers to the primary output store path of the derivation and `name` refers to either:

- The meta.mainProgramattribute of the derivation.
  - The `pname`attribute of the derivation.
- The name part of the value of the`name` attribute of the derivation.

V. Nix Flakes

Flakes are the unit for packaging Nix code in a reproducible and discoverable way. A flake is a filesystem tree often fetched in the form of a git repository or tarball, which contains `flake.nix` file in the filetree's root. `flake.nix` specifies some metadata about the flake such as inputs (the flakes dependencies) and its outputs (Nix values such packages or modules for use in NixOs systems). They can have dependencies on other flakes, making it possible to have multi-repository Nix projects. They have increasingly become the standard for writing and distributing Nix derivations within the community.

All Nix flakes share a common anatomy, they begin with a optional description of the flake and it's declared inputs, which are the dependencies for the flake.

```
description = "A very basic Haskell dev env flake";
inputs = {
    nixpkgs-stable.url   = "github:NixOS/nixpkgs";
    nixpkgs-unstable.url = "github:NixOS/nixpkgs?
ref=nixos-unstable";
};
```

Figure 8. - Input syntax for a Nix flake. Nix is not space sensitive allowing for code alignment for better readabilty.

Flake inputs can be either local or remote, with a differing syntax for each. Remote inputs have a url-like syntax such as: `github:NixOS/nixpkgs`. If the flake input is refers to a local repository, then it will take the form of `git:/home/user/sub/dir`. If the path of the flake input begins with `./` or `/` then it is treated as local path. Inputs in a nix flake can **inherit** from other inputs, which is useful in minimizing flake dependencies, and can simplify any potential debugging. Input inheritance has the following syntax: `inputs.nixpkgs.follows = "dwarffs/nixpkgs";`.

Inputs in a `flake.nix` tend to not have the specific version hashes referenced within the input URL, instead to ensure reproducibility Nix generates a `flake.lock` file in the flakes directory on the first evaluation of said flake. The lockfile is a UTF-8 JSON file containing the graph structure isomorphic to the graph of dependencies for the root flake. An example flake.lock file

Lockfile A type of file generated typically generated by a package manager, it is typically not meant to be edited but is rather a reflection of the resolved dependency tree for a project [6]. They usually have the file type `.lock`


```
{
  "nodes": {
    "nixpkgs-stable": {
      "locked": {
        "lastModified": 1765075010,
        "narHash":
"sha256-8pbe+pDKWWy0qZ4KSz+OtWm2xuLZp4ubPJDuc0TIHi4=",
        "owner": "NixOS",
        "repo": "nixpkgs",
        "rev":
"b12293a0adf2e81e6f6164edd4f4cc8d4e5c17d4",
        "type": "github"
      },
      "original": {
        "owner": "NixOS",
        "repo": "nixpkgs",
        "type": "github"
      }
    },
    "nixpkgs-unstable": {
      "locked": {
        "lastModified": 1764950072,
        "narHash": "sha256-
BmPWzogsG2GsXZtlT+MTcAWeDK5hkbGRZTeZNW42fwA=",
        "owner": "NixOS",
        "repo": "nixpkgs",
        "rev":
"f61125a668a320878494449750330ca58b78c557",
        "type": "github"
      },
      "original": {
        "owner": "NixOS",
        "ref": "nixos-unstable",
        "repo": "nixpkgs",
        "type": "github"
      }
    },
    "root": {
      "inputs": {
        "nixpkgs-stable": "nixpkgs-stable",
        "nixpkgs-unstable": "nixpkgs-unstable"
      }
    }
  },
  "root": "root",
  "version": 7
}
```


Figure 8. - An example `Flake.lock` file, containing 3 nodes and the root flake.

A resolved `Flake.lock` file dependency graph structure is represented through nodes, each node will contain:

- `inputs` - The dependencies of this node, as a mapping from input names (e.g. `nixpkgs`) to node labels.
- `original` -The original input specification from `flake.nix`, as a set of `builtins.fetchTree` arguments.
- `locked` - A set of `builtins.fetchTree` arguments mapped to a specific revision denoted by the hash of the `rev` field.
- `flake` - A Boolean denoting whether this is a flake or non-flake dependency. Corresponds to the `flake` attribute in the `inputs` attribute in `flake.nix`.

Lock files are the key component of reproducible builds and remain static, never changing once resolved. The only way to update the inputs in a nix flake is either manually by editing that inputs' associated revision and hash and changing them to desired version or through the command `nix flake lock --update-input <input-name>` which will update the specified input, `<input-name>`. It is also possible to update all inputs in a flake via the command `nix flake update`, which will generate an entirely new `flake.lock` in-place. If a flake's lock file is deleted, a new one will be generated upon the next evaluation of the flake and dependency versions may differ.

VI. Nix Shells

Setting up complex development environments is incredibly time consuming, and only grows in complexity when taking to account the different configurations individual systems will have. Both Nix and Docker take different approaches to solving this problem.

Docker is a containerisation platform. Containerization allows for applications and their dependencies to be isolated. Each container contains everything need for the application to run, including the application's code/binary, its required libraries and configuration file. Docker containers all share the kernel on the host machine, but isolate the filesystem, networking and application processes. Nix can set up a shell environment called a `nix-shell`, it is a much more lightweight approach to reproducible environments that docker. When load a shell environment, nix evaluations the `shell.nix` or `flake.nix` in the root of the current working directory add loads a shell with the specified packages at their locked versions on path. As apposed to docker, the package dependencies are entirely reproducible. Like Docker, nix-shells do share the host machine kernel but it does not isolate beyond package versions. nix-shells do not permanently install a package, as once you exit a shell, that packages exposed will no longer be available.

```
devShells.${system}.default = pkgs-stable.mkShell {
    packages = [
        pkgs-stable.haskellPackages.haskell-
language-server
        pkgs-stable.cabal-install
        pkgs-unstable.haskell.compiler.ghcHEAD
    ];
}
```

Figure 9. - A nix expression for a dev shell exposing packages for the Haskell language sever, Haskell's compiler GHC and its build system cabal, as part of Flakes outputs.

The function `pkgs-stable.mkShell` is part of Nix's standard library. It a generalised form of the `stdenv.mkDerivation` function and abstracts away some of the common repetition of making shell environments. Other as part of a flake output, nix-shell environments can also be declared using a `shell.nix` file. Nix-shells provide a lightweight alternative to development environments compared to heavier handed Docker. Although there is less isolation of the resulting applications processes, nix-shells require much less resources.

VII. Open source software for the Arts

The appeal of open *Free and Open Source Software* (FOSS) is the increased control over the code running on their machine. As a package and easily be audited by reading through its source code, this level of transparency in turn, increases the security of FOSS software as bugs in the source code can be spotted and patched by that's software's community. FOSS software is very stable for long-term projects due to the distributed nature of the source code. If a FOSS reliant part of program or application grows stale or falls into disrepair by its original creators, then that part of the program can just as easily be continued by community at large and have development continued.

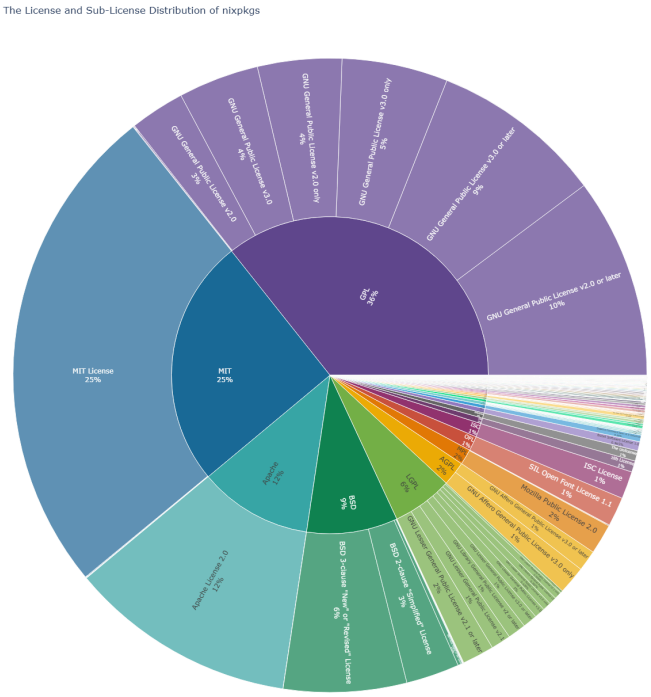

Figure 10. - The license and sub-license distribution in a sample of 16789 different packages available from the nixpkgs repository for the x86_64-linux platform.

Nix is fundamentally built on a source deployment model, where packages are built from source nix deviations into its unique path in the Nix store. For a lot of users, building from source can be frustrating, due the time it can take. However building from source with nix allows for a high level of reproducibility, flexibility and transparency in open source software. By having the compilation of the software taking place on the end users personal hardware a large swath of supply chain vulnerabilities are avoided [7].

VIII. Nixpkgs as a source of free and open source software

The first permissive license, Berkeley Software Distribution (BSD) , was created by University of California Berkeley. The BSD license allows for modification and distribution of the covered software, it comes in either a simplified 2 clause variant, where the non-endorsement clause is omitted or 3 clause variant which prevents the use of the original authors names or trademarks without explicit written permission. In our sample of the nixpkgs repository, 6% of packages use the BSD-3 clause as apposed to 3% using BSD-2. In total with 9%, the BSD license is 4th most used license.

Following the BSD license, the Massachusetts Institute of Technology (MIT) created an open source license based on the original BSD license. The main difference between the MIT and BSD licenses is that MIT does not contain any clauses with regards to promotion or advertising material, but does contain a attribution clause. It is most similar to the BSD-2 license. The MIT license also explicitly allows for the merging, publishing, sublicensing and selling of open source software under its license. 25% of the packages in nixpkgs fall under this license making it the largest single license type in the repository.

Approximately 12% of packages in the nixpkgs repository fall under the Apache License, it the most comprehensive and explicit of the permissive licenses and offers more legal advantages for simple licenses while providing similar grants. The Apache license contains a section on patents, including a patent retaliation clause amongst others.

The GNU General Public License (GPL) is the single largest type of license in the nixpkgs repository, being carried by 36% of the available packages in our sample. Its a reciprocal license that mandates that derivative works must release under a license offering the same freedoms as the GPL license. Like other FOSS licenses it allows for the modification and distribution of the source code. The Lesser General Public License (LGPL) constitutes 6% of the packages in nixpkgs.

With the majority of packages in nixpkgs being openly permissive through the MIT, Apache and BSL licenses (46% combined), licence compatibly of open source components to an artwork remains manageable and flexible with the permissive nature of these licences allowing the creator of the artwork the digression of how they want to handle the source code for their project. Copyleft license are the next largest group in the nixpkgs repository accounting for 42% of the packages in our sample, artworks which that would fall under this license, would have to provide their source code to each institution that hosts said work, however in efforts to reproduce and preserve digital work, supplying artwork with source may ultimately be preferred, especially when utilising the source deployment model which nix thrives upon.

Our sample also contained a small amount closed source and proprietary licenses namely:

- Business Source License 1.1
- Server Side Public License
- Elastic License 2.0
- Functional Source License (all variants)
- Sustainable Use License
- Fraunhofer FDK AAC Codec Library License
- Creative Commons licenses with NC (NonCommercial) or ND (NoDerivatives) clauses
- Apple Public Source License 2.0

Which in total constituted less than 1% of total packages sampled from nixpkgs.

IX. Conclusion

In conclusion, building from source is an excellent way of ensuing software reproducibility and through the use of Nix

Data availability statement

All source data for this project can be found at the projects accompanying repository: . As nixpkgs is set of functions and derivations rather than a static database, to gain a package list for the x86_64-linux platform, the attribute-set containing each package name was first evaluated in the nix repl and each package name was then recorded as a JSON list. Each package was then evaluated to parse its licensing metadata. Each nix expression was ran programactally my calling it as an external process in Rust using `std::process::Command` and the metadata JSON was captured and parsed with the *serde* create. Visualisation was done in Python using the *plotly* library.

Acknowledgements

I would like to acknowledge the many valuable resources that the nix comity and community have made freely available, showing contrary to popular opinion, that Nix is quite well documented. I would also like to pass my gratitude to Patricia Falcao, who was been a wonderful mentor to myself and to express thanks for the opportunity to write this paper.

Bibliography

[1] "Programming - Functions." [Online]. Available: https://users.cs.utah.edu/~germain/PPS/Topics/functions.html

[2] "How Nix Works | Nix & NixOS." [Online]. Available: https://nixos.org/guides/how-nix-works/

[3] "Nix Language Basics — Nix.dev Documentation." [Online]. Available: https://nix.dev/tutorials/nix-language.html#overview

[4] V. J. Illmer, ""Works on My Machine": A Case Study of Replicability Challenges in Computational Humanities Research," *Anthology of Computers and the Humanities*, vol. 3, no., pp. 142–148, Nov. 2025, doi: 10.63744/iaqeoznkfkuz.

[5] "Overlays - NixOS Wiki." [Online]. Available: https://nixos.wiki/wiki/Overlays

[6] Y. Gamage, D. Tiwari, M. Monperrus, and B. Baudry, "The Design Space of Lockfiles Across Package Managers," *arXiv.org*, 2025, doi: 10.48550/arXiv.2505.04834.

[7] P. Dellaiera, "Reproducibility in Software Engineering," *Zenodo (CERN European Organization for Nuclear Research)*, Jun. 2024, doi: https://doi.org/10.5281/zenodo.12666899.